Una extensión para OpenMP que soporta paralelización especulativa

Sergio Aldea, Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano[1]

Resumen— **Las directivas de OpenMP se pueden considerar como el estándar de programación paralela en memoria compartida. Sin embargo, OpenMP no garantiza que la ejecución paralela de un bucle siga la semántica secuencial si aparecen dependencias entre las instrucciones. En este trabajo proponemos aumentar la funcionalidad de OpenMP agregando soporte de paralelización especulativa. Nuestra contribución se resume en tres apartados. Hemos definido un nueva cláusula *speculative* para variables internas de bucles. Esta cláusula asegura que el acceso a estas variables se produce siguiendo la semántica secuencial del bucle. Además, hemos desarrollado una nueva biblioteca de paralelización especulativa que garantiza la ejecución paralela de bucles de OpenMP con variables especulativas. Por último, hemos desarrollado un nuevo paso de GCC que traduce los valores englobados en la cláusula *speculative* en llamadas a la biblioteca especulativa. El resultado es el sistema ATLaS, que utiliza la paralelización especulativa para extender la funcionalidad de OpenMP, y garantizar la semántica secuencial de cualquier bucle ejecutado en paralelo.**

Palabras clave— **Paralelismo, paralelización especulativa, OpenMP, compiladores.**

I. Introducción

EL auge de los procesadores multinúcleo hace necesario profundizar en las técnicas de procesamiento paralelo. Hasta la fecha, se han propuesto lenguajes o extensiones de lenguajes con el objetivo de explotar las características de dichas máquinas. Entre ellas podemos destacar OpenMP [1], una extensión a lenguajes secuenciales como C, Fortran o C++, basada en directivas que permite la ejecución en paralelo de regiones de código definidas por el usuario.

La Fig. 1 muestra un ejemplo de (a) un bucle secuencial en C, y (b) su paralelización con directivas OpenMP. La paralelización de un bucle en OpenMP requiere clasificar las variables de un bucle como private o shared. Informalmente, las variables cuyo valor es definido antes de su uso deben etiquetarse como private, mientras que las variables cuyo valor se comparte durante la ejecución paralela deben clasificarse como shared. En el ejemplo, `a[]` es un vector de sólo lectura compartido, mientras que `v[]` es un vector compartido cuyos elementos se modifican en cada iteración.

OpenMP es un mecanismo simple y potente para paralelizar código; sin embargo, su uso conlleva ciertas limitaciones. Primero, la clasificación de variables es una tarea propensa a errores. Segundo, OpenMP no asegura la ejecución paralela de acuerdo

```
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```
(a)

```
#pragma omp parallel for \
 private (i,b) shared (a,v)
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```
(b)

Fig. 1. Ejemplo de paralelización de un bucle con OpenMP.

```
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```
(a)

```
#pragma omp parallel for \
 private (i,b) shared (a,k) \
 speculative(v)
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```
(b)

Fig. 2. Un bucle que no puede ser paralelizado con seguridad con las cláusulas actuales de OpenMP (a), y su paralelización con la nueva cláusula speculative (b).

a la semántica secuencial, y debe ser el programador el que se ocupe de ello. En el ejemplo visto en la Fig. 1, el programador se encarga de garantizar que cada *thread* modifica un elemento diferente de `v[]`. Tercero, existen muchos fragmentos de código potencialmente paralelizables que no pueden paralelizarse porque el flujo de control depende de datos obtenidos en tiempo de ejecución. Considérese el código representado en la Fig. 2. Supongamos que el valor de `k` no se conoce en tiempo de compilación. Asumiendo que `b>0`, si la ejecución paralela del bucle calcula la iteración `i` *antes* que la iteración `i-b`, los accesos a `v[i-b]` pueden devolver un valor desfasado, rompiendo la semántica secuencial. La única manera de garantizar un comportamiento correcto sería serializar la ejecución de las iteraciones `i-b` e `i`.

Nuestra propuesta consiste en enriquecer OpenMP con una biblioteca que soporte paralelización especulativa, para asegurar que las definiciones y usos de variables compartidas se llevan a cabo de acuerdo a la semántica secuencial de los bucles. Para ello hemos definido una cláusula nueva: speculative. Los accesos a variables etiquetadas como speculative seguirán las siguientes reglas:

- Las lecturas de una variable especulativa devolverán el valor más actualizado posible de la variable. Este valor puede haberse generado previamente por el mismo *thread* o por cualquiera de sus *predecesores* (aquellos *threads* que ejecutan iteraciones previas siguiendo el orden secuencial). Esta operación se denomina *forwarding.*
- Las escrituras de una variable especulativa guardarán el valor en una copia local, y compro-

[1]S. Aldea, A. Estebanez, D. R. Llanos, y A. Gonzalez-Escribano forman parte del Dpto. Informática, Universidad de Valladolid, Campus Miguel Delibes, 47011, Valladolid, Spain. E-mails: {sergio,diego,arturo}@infor.uva.es, palestebanez1@gmail.com

barán si algún *thread sucesor* (los *threads* que ejecutan iteraciones "futuras") ha consumido un valor desfasado de la variable. En cuyo caso, la ejecución del *thread* en cuestión (y posiblemente la de sus sucesores) será detenida y re-ejecutada para forzar la obtención del valor más actual del dato, es decir, aquella que se obtendría mediante una ejecución secuencial. Esta operación se denomina *squash*.

Dado que las violaciones de dependencia fuerzan a descartar los valores de las variables especulativas, todos los *threads* mantienen una versión local de las variables especulativas accedidas. Si un *thread no especulativo* (un *thread* sin predecesores activos) finaliza correctamente la ejecución de su bloque de iteraciones, todos los cambios se consolidan en la versión principal de las variables especulativas.

Las tres principales contribuciones de este artículo son:

1. Hemos definido una extensión de las especificaciones de OpenMP, añadiendo una cláusula que soporta accesos especulativos a datos en regiones `omp parallel for`. Esta cláusula sigue las pautas propuestas por Aldea et *al.* [2].
2. Hemos creado una biblioteca que soporta la ejecución paralela de bucles que incluyen dependencias, incluyendo accesos especulativos a datos basados en punteros de cualquier tamaño sin la necesidad de análisis alguno en tiempo de compilación. Esta biblioteca no solo controla los accesos a datos especulativos, también maneja el reparto de iteraciones entre *threads* y asegura la corrección de la ejecución paralela de un bucle.
3. Finalmente, hemos desarrollado un nuevo paso de compilación a través de un *plugin* en la implementación de GCC OpenMP para dar soporte a la cláusula speculative. Este paso transforma el bucle paralelizado, insertando las llamadas especulativas necesarias para (a) distribuir los bloques de iteraciones entre los procesadores, (b) realizar lecturas y escrituras especulativas en variables, y (c) realizar consolidaciones parciales de los resultados calculados.

El resultado es ATLaS, un *framework* que permite a OpenMP ejecutar bucles en paralelo sin las necesidad de un análisis de dependencias previo. La evaluación de resultados utilizando tanto benchmarks sintéticos, como aplicaciones de propósito específico, en un sistema multicore, muestra que nuestro sistema produce *speedups*.

El resto del trabajo se estructura como sigue. La sección 2 introduce las bases de la paralelización especulativa. La sección 3 describe brevemente la propuesta de una nueva cláusula especulativa para OpenMP. La sección 4 detalla las operaciones de nuestra nueva biblioteca especulativa. La sección 5 muestra la forma de insertar nuestra cláusula especulativa en el compilador de OpenMP de GCC. La sección 6 presenta la evaluación experimental. Finalmente, la sección 7 resume nuestras conclusiones.

II. Paralelización especulativa

La paralelización especulativa, también llamada paralelización optimista [3], asume que el código secuencial puede ser ejecutado de forma optimista en paralelo, mientras un sistema monitoriza la ejecución para asegurar que no se producen violaciones de dependencia. Una violación de dependencia aparece cuando un *thread* genera un dato cuyo valor ha sido consumido por un *thread* sucesor (siguiendo el orden secuencial). En estas situaciones, los resultados calculados por el sucesor son descartados. Las primeras propuestas [4], [5] paraban la ejecución paralela y reiniciaban la ejecución del bucle de forma secuencial. Otras propuestas detienen el *thread* cuyos datos son erróneos, así como sus sucesores, y los re-ejecutaban en paralelo [6], [7], [8], [9]. Una tercera opción (véase por ejemplo [10], [11], [12]) es re-ejecutar solamente el *thread* con datos erróneos, y aquellos *threads* que hayan consumido algún valor del mismo.

La Fig. 3 muestra un ejemplo de paralelización especulativa. En ella se representan cuatro *threads* ejecutando fragmentos de cuatro iteraciones consecutivas del mismo bucle. El valor de `x` no se conoce en tiempo de compilación, por tanto, el compilador no puede asegurar que los accesos a la estructura `SV` no producen violaciones de dependencia en una ejecución paralela. Sin embargo, los valores de `x` para cada iteración se conocen en tiempo de ejecución.

En las ejecuciones especulativas, cada *thread* mantiene una versión local de los datos accedidos especulativamente (aquí, el vector `SV`). En tiempo de compilación, el código original se transforma para realizar lecturas y escrituras especulativas, así como consolidaciones de datos siguiendo el orden secuencial. Los siguientes párrafos describen con más detalle estas operaciones.

Escrituras especulativas. En tiempo de compilación, todas las escrituras realizadas sobre variables especulativas se reemplazan por una función. Esta función escribe el dato en la versión local del *thread* actual, y asegura que ningún *thread* de una iteración posterior ha consumido un valor no actualizado del dato, una situación denominada "violación de dependencia". En caso de que se produzca una violación de dependencia, el *thread* afectado y sus sucesores se paran y reinician. En el ejemplo representado en la Fig. 3, los chequeos en busca de violaciones de dependencia realizados por los *threads* 1 y 2 no encuentran ningún sucesor que haya consumido un dato erróneo de `SV[1]`. Sin embargo, en el instante t_{10}, el *thread* 3 descubre que el *thread* 4 ha utilizado un valor de `SV[2]` desfasado, por tanto, se detecta una violación de dependencia. Por ello, el *thread* 4 debe pararse y reiniciarse. Cuando el *thread* 4 se reinicia, obtiene el valor más actual de `SV[2]` del *thread* 3, continuando la ejecución de las iteraciones asignadas al mismo.

Lecturas especulativas. En tiempo de compilación, todas las operaciones de lectura sobre variables especulativas se reemplazan por una función. Esta función obtiene el valor más actualizado del ele-

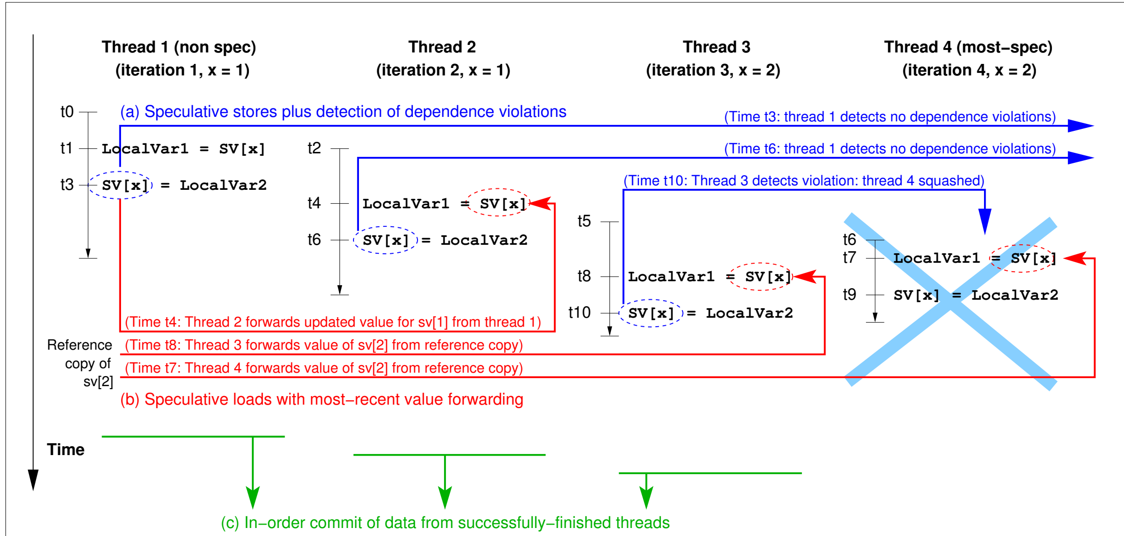

Fig. 3. Ejemplo de la ejecución especulativa de un bucle y resumen de las operaciones que realiza una biblioteca de paralelización especulativa.

mento accedido. Si un predecesor ha leído o modificado el elemento, el valor es recuperado (como hace el *thread* 2 en la Fig. 3). De otro modo, la función obtiene el valor de los datos de referencia (como hace el *thread* 3 en la Fig. 3).

Operaciones de consolidación-o-descarte. Si no se producen violaciones de dependencia, los cambios sobre las variables especulativas se consolidan en la versión de referencia. Esta consolidación debe realizarse en orden para garantizar que se guarda el valor más actualizado posible de los datos. En caso de que haya violaciones de dependencia, los datos intermedios se descartan. En ambos casos, el sistema de reparto de iteraciones asigna un nuevo bloque de iteraciones al *thread* para continuar la ejecución paralela.

Reparto de iteraciones bajo paralelización especulativa. El método utilizado para asignar iteraciones bajo paralelización especulativa es diferente al aplicado en las aproximaciones clásicas, por ejemplo, [13], [14], [15]. En nuestro caso, la ejecución de una iteración o bloque de iteración puede ser descartada, por tanto, el método de reparto debe ser capaz de reasignar el bloque descartado al mismo, u otro *thread*. La estructura del bucle debe modificarse para habilitar dichas re-ejecuciones.

III. Semántica de la cláusula *speculative* en OpenMP

El problema de añadir soporte para paralelización especulativa en OpenMP puede resolverse de dos formas diferentes. La primera de ellas requiere la inclusión de una directiva nueva, como pragma omp speculative for. Sin embargo, hay muchos componentes relativos a OpenMP que debieran ser modificados para agregar una nueva directiva. Una solución más sencilla es añadir una cláusula nueva a la lista de constructores paralelos disponibles, que permita al programador enumerar qué variables deben procesarse especulativamente. La sintaxis de esta cláusula es:

speculative(variable[, lista_var])

Así, si el programador dudase sobre el comportamiento de alguna variable, simplemente la etiquetaría como especulativa. En este caso, todas las definiciones y usos de dicha variable se modificarían con las llamadas a función correspondientes.

Nuestra biblioteca especulativa, descrita en la siguiente sección, se ha desarrollado mediante cláusulas OpenMP. Para integrar dicha biblioteca en una implementación de OpenMP con la cláusula speculative, se tuvieron en cuenta dos particularidades. Primero, dado que nuestra biblioteca se ha implementado utilizando OpenMP, utilizamos variables internas que deben asimismo declararse como private y shared en el bucle en cuestión. Por tanto, si el compilador encuentra una cláusula speculative, además de los cambios citados, deben incluirse las variables de control privadas o compartidas a la lista. Afortunadamente, OpenMP soporta la repetición de cláusulas, lo que permite al compilador añadir cláusulas private o shared adicionales.

Segundo, los métodos estándar de asignación de iteraciones implementados en OpenMP no son suficientes para soportar la paralelización especulativa. Estos métodos asumen que la ejecución de un bloque de iteraciones nunca conduce a errores, no considerando la posibilidad de reiniciar una ejecución por una violación de dependencia. Por tanto, es necesario utilizar un método de asignación apropiado para ejecución especulativa. En lugar de dividir el espacio de iteraciones, hemos adoptado la solución descrita en [6], remplazando la estructura original del bucle por una nueva estructura compuesta de N iteraciones, siendo N el número de *threads*. Al comienzo del bucle, se asigna un bloque de iteraciones diferente a cada *thread*. Si la ejecución es correcta, se asigna un nuevo bloque de iteraciones, de otra forma, el método de asignación intentará reasignar el mismo bloque de iteraciones al mismo *thread*, para mejorar

la localidad y reutilización de cachés. Si no es posible, lo asigna a otro *thread*.

IV. Biblioteca de paralelización especulativa

Hemos desarrollado una biblioteca de paralelización especulativa que soporta la ejecución paralela de bucles for. La arquitectura sigue los principios de diseño de la biblioteca de paralelización especulativa desarrollada por Cintra y Llanos [6], [16]. En estos trabajos, Cintra y Llanos introducen una biblioteca basada en el uso de una ventana deslizante para soportar la ejecución paralela de W bloques de iteraciones consecutivos. Así, cuando el *thread* no especulativo finaliza, sus resultados son consolidados, y el *thread* que ejecuta el siguiente bloque de iteraciones pasa a ser el nuevo *thread* no especulativo. Tras ello, la ventana avanza, permitiendo la ejecución de nuevos bloques de iteraciones. A pesar de su buen rendimiento, la biblioteca desarrollada por Cintra y Llanos sufre de ciertas limitaciones. Primero, requiere que todas las variables especulativas sean almacenadas en un vector único antes de la ejecución del bucle. Segundo, todas las variables especulativas deben compartir un mismo tipo de datos. Tercero, sólo podemos acceder a las variables por nombre (no se pueden referenciar por dirección o punteros). Finalmente, esta biblioteca crea W copias de la estructura de datos especulativa, siendo W el tamaño de la ventana deslizante utilizada, en lugar de guardar solamente las diferentes versiones de los datos accedidos. Estas limitaciones impiden que esta biblioteca pueda utilizarse para dar soporte a la cláusula speculative, donde las variables y estructuras de datos etiquetadas como especulativas pueden ser de diferentes tipos de datos, accedidas por nombre y/o dirección, y donde las estructuras de datos especulativas pueden ser de cualquier tamaño.

Nuestra nueva biblioteca especulativa supera estas limitaciones, lo que permite acceder especulativamente a variables de cualquier tipo de datos, por nombre y/o dirección, y controlar bajo demanda el espacio necesario para las versiones locales.

A. Lecturas y escrituras especulativas

La interfaz de nuestra implementación de la función de lectura especulativa es como sigue:

```
specload(VOID* addr, UINT size, UINT chunk_number, VOID* value)
```

El primer parámetro es la dirección de la variable especulativa; el segundo es el tamaño de la variable; el tercero es el número de bloque de iteraciones ejecutado por el *thread* (necesario para deducir el *slot* utilizado); y el cuarto es un puntero al lugar donde se devolverá el dato requerido. Cabe destacar que specload() debe devolver el valor disponible más actualizado posible de la variable especulativa.

La interfaz de la función de escritura especulativa se denomina specstore() y es similar a la de specload(), con la salvedad de que en este caso, el último parámetro es un puntero al valor que debe ser almacenado. Cabe mencionar que specstore() no sólo debe guardar el nuevo valor, además debe comprobar si un sucesor ha consumido un valor desfasado de la variable en cuestión.

B. Operación de consolidación parcial

La operación de consolidación parcial la realiza exclusivamente el *thread* no especulativo. Cada vez que un *thread* ejecuta la función commit_or_discard(), primero comprueba si su trabajo no debe ser descartado debido a una violación de dependencia, y si se trata del *thread* no especulativo. Si el *thread* es especulativo, se marca el *slot* como "terminado" para que sea consolidado por el *thread* no especulativo cuando llegue el momento.

Cabe destacar que cada *thread* sólo escribe en su versión local de datos, por tanto, no es necesario proteger las operaciones con secciones críticas, reduciendo al mínimo el uso de las mismas.

La biblioteca descrita puede verse con más detalle en [17], [18].

V. Soporte de compilación para la cláusula *speculative*

La fase de compilación de nuestro sistema se implementa sobre el compilador GCC [19], extendiendo su funcionalidad a través de un *plugin*.

La arquitectura de GCC. La Fig. 4 muestra el esquema de la arquitectura de GCC [20], [21]. A grandes rasgos, GCC convierte la representación de un programa en otra, a través de diferentes pasos. Cada paso genera un representación de nivel más bajo, hasta llegar en el último paso, a código ensamblador.

Soporte para la nueva cláusula. Se ha extendido el compilador de GNU OpenMP (GOMP). Las partes más relacionadas con OpenMP se destacan en gris en la Fig. 4. En este trabajo nos hemos centrado en modificar los pasos de compilación donde se genera el código intermedio común para todos los lenguajes, así el nuevo código se genera mediante el *plugin* desarrollado, que consiste en llamadas a las funciones descritas en la sección anterior. Podemos ver la localización del *plugin* destacado en negro en la Fig. 4.

El analizador sintáctico identifica las cláusulas y directivas de OpenMP, y produce un código genérico. Hemos modificado el analizador sintáctico de C y la representación generada con el fin de dar soporte a la nueva cláusula speculative. Cuando nuestro *plugin* detecta la cláusula, realiza las transformaciones necesarias al código en el código fuente.

Descripción del *plugin* especulativo de GCC. Los *plugins* de GCC se encargan de proporcionar al compilador nuevas características[1] permitiendo añadir, remplazar, monitorizar o incluso eliminar pasos del compilador GCC sin tocar el código original de GCC. Nuestro trabajo se basa en agregar un nuevo paso a las transformaciones de GCC que modifica el código cuando un programa contiene la cláusula speculative.

[1]Sin embargo, no pueden extender el lenguaje analizado

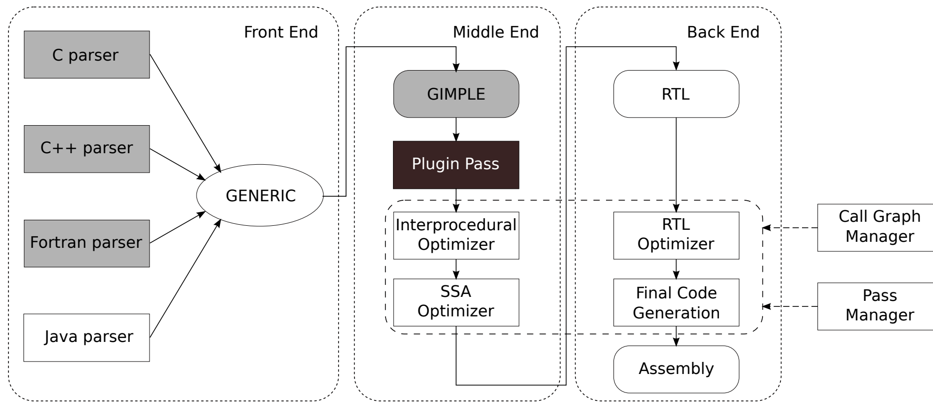

Fig. 4. Arquitectura del compilador GCC. Los principales componentes relacionados con OpenMP, destacados en gris, son los analizadores sintácticos de C, C++ y Fortran, y el nivel GIMPLE IR. En negro se destaca la localización de nuestro *plugin*.

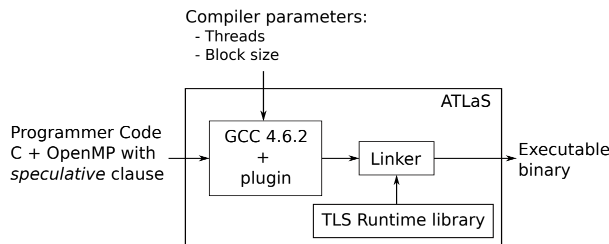

Fig. 5. Visión general del proceso de generación de código de la cláusula speculative

Este nuevo paso se añade antes de las fases de optimización del compilador, y justo antes del paso en que el compilador se relaciona con OpenMP. Así, tratamos con el código en una forma en que las directivas de OpenMP se conservan, y ya disponemos de información de las variables marcadas como privadas, compartidas o especulativas. Tras este paso, GCC trata las variables especulativas como compartidas, mientras que el tratamiento especulativo se lleva a cabo por la biblioteca especulativa descrita anteriormente. De este modo, el analizador, al detectar una cláusula speculative, reemplaza las lecturas y escrituras sobre variables especulativas por las correspondientes llamadas a funciones. Además, inserta el código necesario para repartir las iteraciones y re-ejecutar en caso de violaciones de dependencia. Este proceso es completamente transparente para el programador, que no necesita saber nada sobre el modelo de programación especulativo. El programador sólo necesita etiquetar las variables del bucle objetivo como private o shared, como en otros programas de OpenMP, y marcar como speculative aquellas variables que puedan dar lugar a violaciones de dependencia.

Uso de ATLaS. Para paralelizar un bucle con nuestro sistema, los programadores sólo deben añadir la directiva de OpenMP y clasificar las variables del bucle objetivo de acuerdo a su uso como private (y sus variantes), shared, o speculative. Para compilar el programa, también debe indicarse el tamaño del bloque de iteraciones que debe utilizarse en la ejecución especulativa, junto con otros parámetros menos significativos. La Fig. 5 resume el proceso de generación de código realizado por el *plugin*, y el enlace con la biblioteca de paralelización especulativa, transparente al usuario.

VI. Evaluación experimental

Los experimentos se llevaron a cabo en un servidor con 64 procesadores, equipado con cuatro chips de 16 cores AMD Opteron 6376 a 2.3GHz, con 256GB de RAM, que utiliza Ubuntu 12.04.3 LTS. Todos los *threads* tenían acceso exclusivo a los procesadores durante la ejecución de los experimentos. Para las medidas se ha utilizado tiempo absoluto. Hemos utilizado el *plugin* ATLaS junto con gcc para todas las aplicaciones. Los tiempos mostrados representan el tiempo de ejecución de los bucles paralelos para cada aplicación, por tanto, no se han tenido en cuenta los tiempos para leer los conjuntos de entrada, ni para mostrar los resultados.

A. Aplicaciones utilizadas

Para probar el sistema desarrollado hemos utilizado tanto benchmarks sintéticos, como aplicaciones de propósito específico. Estas aplicaciones incluyen el cálculo del menor círculo contenedor en dos dimensiones (2D-MEC), el cálculo del cierre convexo en dos dimensiones (2D-Hull), el problema de la triangulación de Delaunay, y una implementación en C del TREE.

La Fig. 6 muestra los *speedups* alcanzados utilizando la cláusula propuesta speculative con las aplicaciones de propósito específico. Para el benchmark 2D-MEC, nuestra solución alcanza un pico de 2.6×. Los resultados del 2D-Hull dependen del conjunto de entrada. Así, los rendimientos varían desde un *speedup* de 2.4× con un conjunto de entrada en forma de *Disco*, que causa un número elevado de violaciones de dependencia, a un *speedup* de 13× con el conjunto de entrada que sigue una distribución de *Kuzmin*, que produce menos violaciones de dependencia. La ejecución de Delaunay produce un gran número de violaciones de dependencia, afectando en gran medida al *speedup*. Aún así, se alcanza un *speedup* pico de 3.1×. Finalmente, el TREE obtiene un pico de *speedup* de 6.5×. Este benchmark se caracteriza por la presencia de reducciones sobre la suma y el máximo en las variables especulativas del bucle a para-

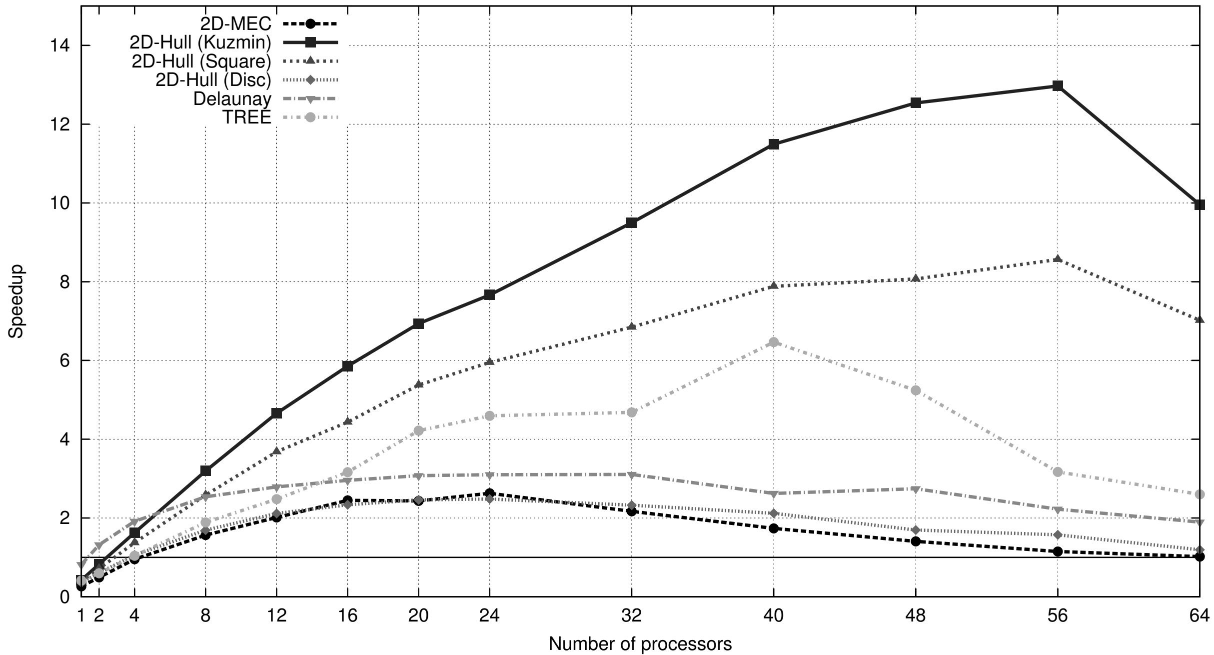

Fig. 6. Rendimiento de diferentes aplicaciones.

lelizar.

B. *Benchmarks sintéticos*

Además de más de 75 test de regresión, se han desarrollado tres benchmarks sintéticos. Uno de ellos comprueba que nuestra solución es genérica, paralelizando especulativamente variables de diferentes datos escalares, con diferentes tamaños, así como elementos de estructuras más complejas. El segundo realiza operaciones de forma exhaustiva (una lectura y una escritura sobre la misma variable en todas las iteraciones) para comprobar que la solución es robusta. Por último, el tercer benchmark comprueba la eficiencia de nuestro sistema. Cabe mencionar que los primeros realizan sus tareas más despacio que en su respectiva versión secuencial, pero de forma correcta. El benchmark que prueba la eficiencia del sistema sólo produce dos violaciones de dependencia en sus 180 000 iteraciones (este hecho es suficiente para que no se pueda paralelizar en tiempo de compilación). Ejecutando este benchmark en nuestro sistema obtenemos un *speedup* máximo de 44.5× con 64 procesadores, mostrando una eficiencia de más del 90% cuando se dedican hasta 32 procesadores para dicha tarea. Por tanto, podemos afirmar que la sobrecarga producida por la biblioteca especulativa es mínima.

VII. Conclusiones

ATLaS permite paralelizar fácilmente bucles que no pueden analizarse en tiempo de compilación y/o presentan dependencias en una posible ejecución paralela. La solución se basa en el uso de una nueva cláusula speculative para marcar las variables que pueden producir violaciones de dependencia. El uso de esta solución no requiere más conocimiento que el necesario para utilizar las directivas estándar de OpenMP. Además, simplifica la tarea de clasificación de variables de acuerdo a su uso: si un programador no está seguro de la posibilidad de paralelizar un bucle, puede etiquetar las variables que le hacen dudar como speculative. Esta decisión garantiza la ejecución correcta del bucle en paralelo, posiblemente a costa de un rendimiento más bajo. Además de respetar la semántica secuencial de los bucles, nuestra solución también alcanza *speedups* notables en aplicaciones que contienen bucles no paralelizables en tiempo de ejecución.

Nuestro trabajo futuro plantea diferentes vías como explorar otros mecanismos de reparto de iteraciones distintos de la asignación fija implementada actualmente, o estudiar otras alternativas a la hora de descartar los resultados cuando se producen violaciones de dependencia, que permitan maximizar la reutilización de resultados.

Agradecimientos

Este trabajo ha sido financiado parcialmente por la Junta de Castilla y León (VA172A12-2, PIRTU); Ministerio de Industria, España (CENIT OCEAN-LIDER); MICINN (España) y la Unión Europea FEDER (proyecto MOGECOPP TIN2011-25639, Red CAPAP-H4 TIN2011-15734-E).

Referencias

[1] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, 1 edition, Oct. 2000.

[2] Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano, "Support for thread-level speculation into OpenMP," in *IWOMP'12 Proceedings*, June 2012, pp. 275–278.

[3] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew, "Optimistic parallelism requires abstractions," in *PLDI'07 Proceedings*, 2007, pp. 211–222.

[4] Manish Gupta and Rahul Nim, "Techniques for speculative run-time parallelization of loops," in *SC'98 Proceedings*, 1998, pp. 1–12.

[5] Lawrence Rauchwerger and David Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI'95 Proceedings*, 1995, pp. 218–232.

[6] Marcelo Cintra and Diego R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *PPoPP'03 Proceedings*, June 2003, pp. 13–24.

[7] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *IPDPS'02 Proceedings*, 2002, pp. 20–29.

[8] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra, "Combining thread level speculation helper threads and runahead execution," in *ICS'09 Proceedings*, 2009, pp. 410–420.

[9] Lin Gao, Lian Li, Jingling Xue, and Pen-Chung Yew, "Seed: A statically-greedy and dynamically-adaptive approach for speculative loop execution," *IEEE Transactions on Computers*, vol. 62, no. 5, pp. 1004–1016, May 2013.

[10] Xiao-Feng Li, ZhaoHui Du, Chen Yang, Chu-Cheow Lim, and Tin-Fook Ngai, "Speculative parallel threading architecture and compilation," in *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, Washington, DC, USA, 2005, ICPPW '05, pp. 285–294, IEEE Computer Society.

[11] Chen Tian, Min Feng, and Rajiv Gupta, "Speculative parallelization using state separation and multiple value prediction," in *Proceedings of the 2010 International Symposium on Memory Management*, New York, NY, USA, 2010, ISMM '10, pp. 63–72, ACM.

[12] Álvaro García-Yágüez, Diego R. Llanos, and Arturo Gonzalez-Escribano, "Squashing alternatives for software-based speculative parallelization," *IEEE Transaction on Computers*, to appear.

[13] Torben Hagerup, "Allocating independent tasks to parallel processors: An experimental study.," *J. Parallel Distrib. Comput.*, vol. 47, no. 2, pp. 185–197, 1997.

[14] C.P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, 1985.

[15] Ten H. Tzen and Lionel M. Ni, "Trapezoid self-scheduling: A pratical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, 1993.

[16] Marcelo Cintra and Diego R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 562–576, 2005.

[17] Alvaro Estebanez Lopez, Diego R. Llanos, and Arturo Gonzalez-Escribano, "Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros," in *Proceedings of the XXIII Jornadas de Paralelismo*, Elche, Alicante, Spain, September 2012.

[18] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano, "Improving the perfomance of a pointer-based, speculative parallelization scheme," in *Proceedings of the 1st First Congress on Multicore and GPU Programming*, February 2014, PPGM'14.

[19] GNU Project, "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, [Last visit: March 2014].

[20] GNU Project, "GCC internals," http://gcc.gnu.org/onlinedocs/gccint/, [Last visit: March 2014].

[21] Diego Novillo, "GCC an architectural overview, current status, and future directions," in *Proceedings of the Linux Symposium*, Tokyo, Japan, September 2006, pp. 185–200.